A Transformation Approach for Multiform Time Requirements

Nadia Menad[1] and Philippe Dhaussy[2]

[1] University of Science and Technology
of Oran Mohamed Boudiaf, Algeria
[2] UEB, LabSticc Laboratory UMR CNRS 6285
ENSTA Bretagne, France
firstname.name@ensta-bretagne.fr

Abstract. Many of the timing constraints expressed in physical prescriptions of distributed systems and multi-clock electronic systems can be expressed in logical concepts. A logical time model has been developed as a part of the official OMG UML profile MARTE, in order to enrich the formalism of this profile and also to facilitate the description and analysis of temporal constraints.

This time model is associated with CCSL (Clock Constraint Specification Language). Once the software is modeled, the difficulty lies in both expressing the relevant properties and in verifying them formally. We present an automatic transformation technique related to a method for verifying properties by model checking, thus exploiting both the CDL language (Context Description Language) and the OBP tool (Observer-based Prover). The technique is based on a translation of MARTE models and the CCSL constraints into Fiacre code. CDL can express predicates and observers. These are verified during the exhaustive exploration of the complete model by OBP. We illustrate our contribution by an illustrative case.

Keywords: Formal verification, model-checking, CCSL time constraints, observer automata.

1 Introduction

In the field of modeling software architectures of distributed systems, control-command systems or multi-clock electronic systems, the specification of functional parts of systems is often associated with temporal constraint specifications. These systems are often critical and the requirements to be respected during the modeling step, concern not only the determinism but also temporal constraints at a functional level. In the system development process, the designers look for methods and languages that allow them to describe their architectures, throughout the cycle and at various levels of abstraction. At each level, the modeling of such systems should allow the expression and the manipulation of time requirements, and the evaluation of the accuracy and efficiency of applications in terms of temporal and measurable requirements.

R.M. Hierons, M.G. Merayo, and M. Bravetti (Eds.): SEFM 2013, LNCS 8137, pp. 16–30, 2013.

For this purpose, the concept of abstract modeling of logical clocks has been introduced with the CCSL language (*Clock Constraint Specification Language*) [And09] within MARTE [MAS08] and adopted by the OMG [OMG10]. CCSL is a language to define causal, chronological and temporal relationships. It aims to complement the existing formalisms and to provide models which can be analysed so as to assess their accuracy with regard to requirements expressed by the designer. It is therefore essential to adopt temporal analysis approaches by integrating verification and validation processes based on robust formal notions, in order to meet current quality requirements of these systems.

To address this issue, several studies have proposed an engineering approach founded on models, and the use of semi-formal notations such as UML, enriched with formal notations. For example, UML-MARTE profile aims to express temporal constraints on UML models. The models that are built must not only be simulated but also interpreted during formal analysis so as to check the temporal requirements defined by the designer. In this study, we use *model-checking* verification techniques [QS82, CES86]. These techniques have become highly popular due to their ability to confirm software model properties automatically.

This paper describes exploratory work which studies the association of CCSL constraint specification and a formal property verification tool named OBP (Observer-Based Prover)[1] [DBRL12]. The verifications carried out by OBP are based on the exploration of Fiacre programs [FGP+08] as well as the exploitation of observers (Fig. 1). The OBP imports Fiacre models corresponding to a translation of UML-MARTE models including CCSL specifications. In addition, it imports CDL programs which describe the properties and context scenarios if required. OBP explores the model and evaluates, at each step of the running model, the value of predicates and the status of all involved observers. Through this approach, we endeavor to verify both functional and temporal properties of programs by combining CCSL constraints with the modeled software architecture.

Our contributions are as follow: (1) we generate Fiacre programs from UML-MARTE; (2) we exploit the CCSL specifications and enrich these programs by the addition of Fiacre constraint processes implementing CCSL, taking inspiration from the approach described in [YM11]. We describe, in this paper, the principles of the Fiacre code generation from CCSL constraints; (3) we show how to specify observer automata exploiting CDL and to use the OBP tool to verify them based on generated Fiacre code; (4) we illustrate our contribution with an example and describe the results of the proofs of the requirements conducted.

This paper is organized as follows: Section 2 presents related work in formal analysis and verification of CCSL constraints. We present the CCSL language in Sect. 3. An illustrative case study is presented in Sect. 4 and the principles of transformation of CCSL constraints into Fiacre are introduced in Sect. 5. Section 6 describes the verification technique based on observers and, in Sect. 7, we introduce and discuss some results of property proofs. We conclude in Sect. 8.

[1] `http://www.obpcdl.org`

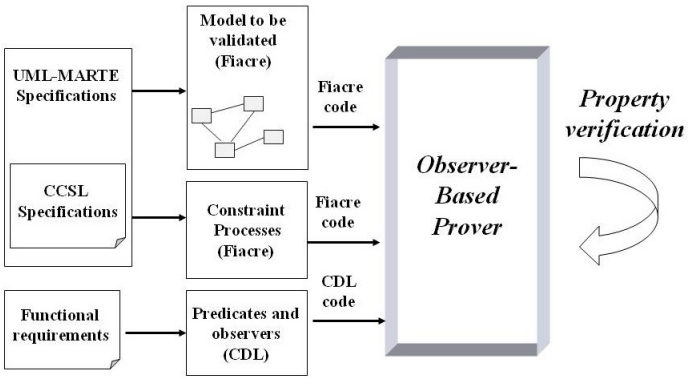

Fig. 1. OBP tool for verification

2 Related Work

Many studies have been conducted to formally verify CCSL constraints. For instance, the approach [YTB+11] presents an extension in response to CCSL specifications. The paper suggests a framework for translating CCSL specifications in dynamical systems, which are handled using the Sigali model-checker to apply the satisfaction of specified constraint relations. However, this approach is too restrictive because it only focuses on the implementation of CCSL constraints with Signal. [And10] proposed an approach for implementing observers [HLR93] for the formal verification of CCSL specifications. Observers, encoding CCSL constraints are translated into Esterel code. [Mal12] describes a technique to generate VHDL code from a CCSL specification. In these approach, a reachability analysis allows to determine whether an observer has reached an error state. The Times Square Environment [DMA08], dedicated to solving CCSL constraints and computing solutions, implements a code generator in Esterel. In contrast to these works, we propose a more general translation approach that verifies not only CCSL constraints implementation, but also properties on the complete model including all the functional components. Furthermore, in our approach these properties are separated from application model thanks to our CDL language, thus separating concerns.

[YM11] proposes a translation of CCSL specifications into a Promela model to formally verify the CCSL constraints by the SPIN model checker. We have been inspired by this work to design the automatic translation of CCSL constraints into Fiacre automata. Also, in this approach the properties to be checked are expressed in LTL logic. We propose to express properties with the CDL language with observer automata which allow a better expressiveness. For example, in our paper, we show a property (illustrated in Fig. 7) that would be tedious to express in LTL.

3 The CCSL Modeling

CCSL, introduced as an annex of MARTE, is a declarative language used to specify binary relations between events based on logical clock concepts. In a MARTE model, any event (for example a communication, transmission or reception action, as computing start) may be used to define a time base, considered to be a logical *clock*. A *clock* represents a set of occurrences of discrete events, called *instants*. These instants are strictly ordered and provide a temporal reference. We briefly recall below some examples of CCSL constraints.

3.1 Examples of CCSL Constraints

We present here some of the relations described in [And10, YM11], which are necessary for the model implementation of the illustrative case study described in this article, namely the relation of alternative, strict precedence and filtering.

An **alternative relation** (denoted $alternatesWith$) is a relation between two asynchronous clocks C_1 and C_2. It specifies that for any natural number k, the k^{th} instant of C_1 occurs before the k^{th} instant of C_2, and the k^{th} instant of C_2 occurs before the $k+1^{th}$ instant of C_1. For our case study, we illustrate the relation $write_i$ $alternatesWith$ $read_i$ by the chronogram in Fig. 2.a and the automaton in Fig. 2.b. Note that for the non-strict alternation in the expression (1) above, the symbol $\prec$ must be replaced by $\preceq$.

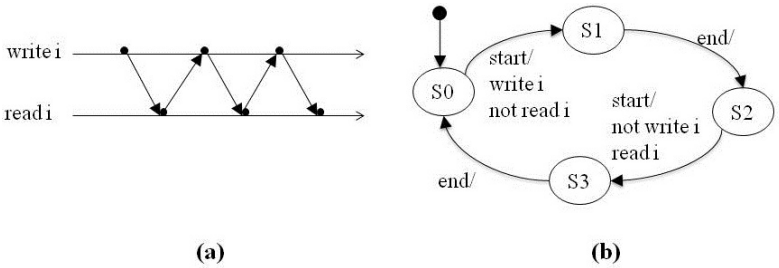

Fig. 2. Illustration of the alternation constraint : $write_i$ alternatesWith $read_i$

A **precedence relation** (denoted $strictPrec$) is an asynchronous relation between two clocks C_1 and C_2. C_1 is said to be strictly faster than C_2, where "C_1 strictly precedes C_2", noted C_1 $strictPrec$ C_2, specifies that for any natural number k, the k^{th} instant of C_1 occurs before the k^{th} instant of C_2, $i.e$ $\forall k \in N^*,\ C_1[k] \prec C_2[k]$.

A **filter relation** (denoted $filteredBy$) is a relation which defines a sub-clock from a given discrete clock. The mapping between the two clocks is characterized by a *filtering pattern* (or simply filter) encoded by a finite or infinite binary word $w \in \{0,1\}^* \cup \{0,1\}^w$. C_1 $filteredBy$ w, defines the sub-clock C_2 of C_1 such as $\forall k \in N^*,\ C_2[k] \equiv C_1[w \uparrow k]$, where $w \uparrow k$ is the index of the k^{th} 1 in the pattern w. The binary words are used to represent sequences of activations. When the latter are periodic, they can be represented by periodic binary words denoted

by $w = u(v)^w$. u and v are finite binary words, called respectively prefix and period.

4 Illustration through a Simple Case Study

We consider a data acquisition circuit (C), with two channels, consisting of acquisition components ($Sensor_i$ and Acq_i) ($i \in \{1, 2\}$), an acquired data processing component ($Comput$) and a filter ($Filter$) sampling the calculated values. Each acquisition channel i is associated with a pair of components $Sensor_i$ and Acq_i. We assume that, for each channel i, the component $Sensor_i$ receives data from the environment (from a device Dev_i outside the circuit) and transmits the value to Acq_i through a shared memory M_i. Each Dev_i sends N data $data_{ik}, k \in [0 \dots N-1]$. Acq_i provides $Comput$ with each datum $data_{ik}$ via a synchronous communication port $portAcq_i$. $Comput$ applies the addition of $data_{1k}$ and $data_{2k}$ respectively received from Dev_1 and Dev_2 and provides the $Filter$ with the sum via a $fifo$. $Filter$ provides the sampled data (one in every three values) to Dev_{out}, external to the circuit.

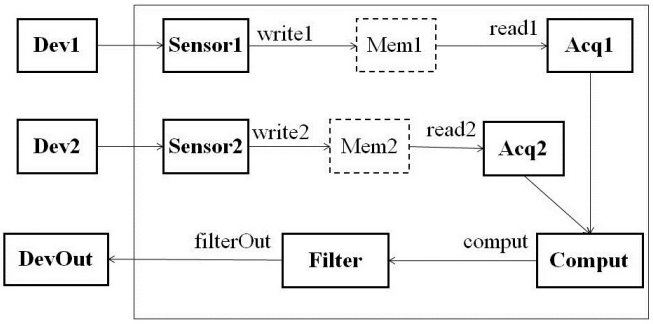

Fig. 3. Circuit architecture C

The temporal constraints associated with this circuit are:

- $Req1$: Each acquired datum $data_i$ is written in the memory M_i before being read by Acq_i (with $i \in \{1, 2\}$).
- $Req2$: $Comput$ starts the calculation of a sum after two receptions of $data_ik$ from each Acq_i (with $i \in \{1, 2\}$).
- $Req3$: $Filter$ provides the environment with a sampled value from a sequence of one in every three values calculated by $Comput$.

In summary, all the timing requirements for our case study, are specified with CCSL language as follows :

$$write_1 \ alternatesWith \ read_1 \qquad (Req1)$$
$$write_2 \ alternatesWith \ read_2 \qquad (Req1)$$
$$read_1 \ strictPrec \ comput \qquad (Req2)$$
$$read_2 \ strictPrec \ comput \qquad (Req2)$$
$$filterOut \ = \ comput \ filteredBy \ (001)^w \ (Req3)$$

In addition to the above time constraints, we express the requirements that are specifically associated with the expected behavior of the circuit. For example, we can express the following requirement:

– *Req*4 : the data $result_j, j \in [0 \ldots (N-1)/3]$ provided to the environment after the sampling operation (one value in 3) must have the values $data_{1k} + data_{2k}$ *with* $k = (3 * j) + 2$.

5 Translation Principles of the CCSL Constraints into Fiacre Programs

This section presents the concepts of Fiacre programs and the translation principles of CCSL constraints into Fiacre programs. These principles have been implemented in our code generator.

5.1 The Fiacre Language

The Fiacre language (*Intermediate Format for the Architectures of Distributed Embedded Components*) has been developed within the TOPCASED project[2] as a key language linking high-level formalisms such as UML, AADL and SDL with formal analysis tools. Using an intermediary formal language has the advantage of reducing the semantic gap between the high-level formalisms and the descriptions internally manipulated by verification tools such as Petri nets, process algebras or timed automata. Fiacre is a language with a formal semantic that serves as input language for different checking tools. Fiacre allows the behavioral and timed aspects of real-time systems to be described. It integrates the notions of process and components as follows:

– the processes (*process*) describe automata with a set of states and a list of transitions between these states. These later reference classical operations (variable allocations, if-elsif-else, while, sequence compositions), non-deterministic constructions and communications done via ports and via shared variables;
– the components (*component*) describe compositions of processes. A system is built as a parallel composition (clause *par* with the || operator) of components and/or processes that can communicate via ports. The Fiacre processes can be synchronized with or without value passage via the ports. They can also exchange data via asynchronous communication queues using shared variables.

5.2 Translation Principles

The general idea of the translation is based on (1) the generation of a Fiacre *Scheduler* process, (2) the generation of Fiacre processes corresponding to the

[2] `http://www.topcased.org`

CCSL constraints and (3) the generation of Fiacre component. The principles of translating CCSL constraints into Fiacre programs and the generation of *Scheduler* code are inspired by the work described in [YM11]. We suppose here that the active objects of the UML model are generated into Fiacre processes with a translation which is not detailed in this paper.

The role of the *Scheduler* process is to determine the order of execution of functional processes based on the constraint process state. *Scheduler* is in charge of activating each functional process. To do so, *Scheduler*, the constraint processes and the functional processes are all synchronized through (synchronous) communication ports. Figure 4 illustrates partially the generation of code for $Sensor1$, $Acq1$ and the $alternatesWith$ constraint. In this figure, we illustrate the synchronization links with dash lines. For example, $Sensor1$ is synchronized with *Scheduler* via the port $sync\_pw1$ to execute a writing operation of a given datum *data* in memory $M1$ shared between $Sensor1$ and $Acq1$ processes. $AlternatesWith$ process is synchronized with *Scheduler* via the ports $startA1$, $updateA1$ and $endA1$.

$Acq1$ and $Comput$ communicate through port $portAcq1$ with a integer value. $Comput$ and $Filter$ communicate through a shared variable $fifoFromComput$ of $fifo$ type. $Filter$ is synchronized with *Scheduler* via $sync\_filter$ for filtering operation. $sync\_filter$ carries a boolean value needed by the $Filter$ behavior. The *Scheduler* process and constraint processes share logical clocks (table tab_Clocks) that correspond to events occurring in the circuit computation ($write1$, $write2$, $read1$, $read2$, $comput$, $filterOut$). The same translation process is applied to other functional processes $Sensor2$, $Acq2$, $Comput$, $Filter$ and the other constraint processes $StricPrec$ and $FilteredBy$.

For this case study, we implement the objects Dev_1, Dev_2 and Dev_{out} with the CDL language, because we consider that these objects run in the environment of the circuit[3].

Generation of a Fiacre Component: The Fiacre program includes a component called C (cf Listing 1) that contains the instances of the processes running at the same time (operator ||). As result of generation algorithm execution, the codes of functional processes, constraint processes and Scheduler are generated. The functional processes are generated from active objects of the UML model and correspond to the functional parts of the model.

For automatic code generation is possible, we must declare clock numbers and links between clocks and synchronization triggers generated by *Scheduler*. For example, the clock $read1$ is associated with $sync\_pr1$ synchronization port to synchronize the first instance (*Acq:1*) of *Acq* process. The clock $filter$ is associated with $sync\_filter$ synchronization port which carries a boolean value. For this last constraint, in our implementation, we need two indices in the table tab_Clocks. These attributes are specified as follows:

[3] The description of CDL language can be found at `http://www.obpcdl.org`

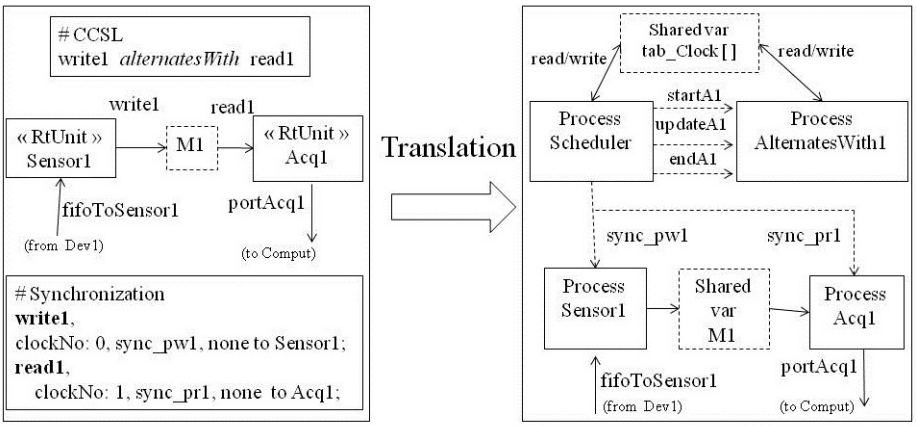

Fig. 4. Illustration of the Fiacre architecture partially generated

```
# Synchronization
write1: clockNo: 0, synchro: sync_pw1 none  to:  Sensor:1;
read1:  clockNo: 1, synchro: sync_pr1 none  to:  Acq:1;
write2: clockNo: 2, synchro: sync_pw2 none  to:  Sensor:2;
read2:  clockNo: 3, synchro: sync_pr2 none  to:  Acq:2;
comput: clockNo: 4, synchro: sync_comput none  to: Comput:1;
filterOut: clockNo: 5, synchro: sync_filter bool:true,
           clockNo: 6, synchro: sync_filter bool:false to: Filter:1;
```

In our case study, the code generator produces 12 processes: *Scheduler*, 5 constraint processes (2 for *alternatesWith*, 2 for *strictPrec*, 1 for *filterBy*) and 6 functional processes (*Sensor*1, *Sensor*2, *Acq*1, *Acq*2, *Comput* and *Filter*). The Fiacre code of the partial component part is generated as follows[4]:

```
component C is
var write1, read1, ..., M1 : int, tab_Clocks : T_ARRAY_CLOCK,
    fifoToSensor1, fifoFromComput : fifo, ...
port startA1, sync_pr1, sync_pw1, ... : none, portAcq1: int, ...
init write1 := 0; read1 := 1; ...        // clock numbers
par
//-------- Scheduler process ---------
 Scheduler [startA1, ... sync_pr1, sync_pw1, ...] (&tab_Clocks)
 //-------- constraint processes ---------
 || AlternatesWith [startA1, ...](&write1, &read1, &tab_Clocks)
 || ...
 //-------- functional processes ---------
 || Sensor1 [sync_pw1] (&fifoToSensor1, &M1)
 || Acq1 [sync_pr1, portAcq1] (&M1)
 || ...
end C
```

Listing 1. Partial generated component program

[4] The complete code of the case study can be found on site `http://www.obpcdl.org`

Generation of Scheduler: The principle of the *Scheduler* process execution is as follows: for each iteration, It executes a number of steps as shown (Fig. 5.a): (1) the *Start* step for the declared clocks initialization and the activation of constraint processes. (2) the *End* step for the synchronization at the end of the constraint processes. (3) An active phase during which the *Scheduler* synchronizes with each functional process so that each process runs. A execution period corresponds to the time between two *start* steps. (4) An intermediate phase *Update* is interposed between the *start* steps and *end* steps to synchronize some constraints if required. The algorithm executed by *Scheduler* is repeated to simulate the coincident moment sequence (an *instant*). Interleaving or simultaneous execution of functional processes is simulated by synchronization between *Scheduler* and the functional processes involved, at every temporally bounded instants. For example, Fig. 5.b shows two clocks $ck1$ and $ck2$ that are activated in each case at the same time. $ck3$ alternates with $ck1$ or $ck2$.

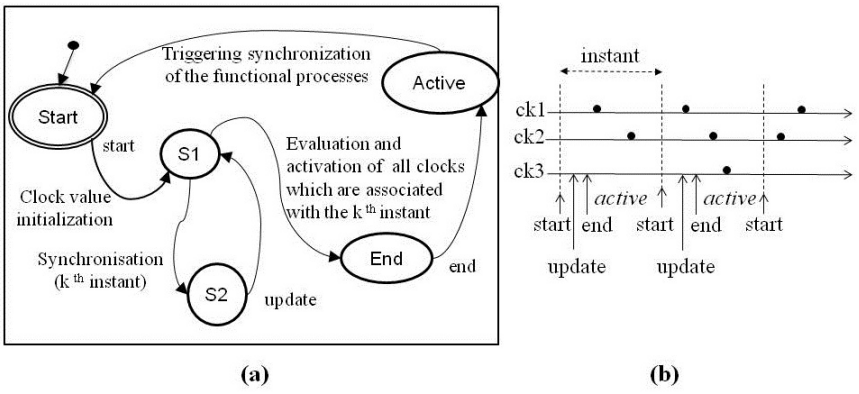

Fig. 5. Scheduler process automaton

From the point of view of the Fiacre implementation, and taking into account the Promela program implementation principle described in [YM11], each event in the model gives rise to a *clock* which is located by a Fiacre structure *tab_Clocks*. This structure is declared as follows:

```
type T_CLOCK is record clock_state:nat, enable_tick, dead: bool end
type T_ARRAY_CLOCK is array 7 of T_CLOCK
tab_Clocks: T_ARRAY_CLOCK
```

In each iteration of the *Scheduler*, each constraint process updates value *clock_state* which takes integer values 0, 1 or 2, in accordance with the execution of the automaton it encodes. Once the process has executed a loop constraint, *Scheduler* evaluates these values to set the value *enable_tic* to *true* or $false$. If *enable_tic* is evaluated as *true*, the functional process associated with the event is synchronized with *Scheduler*, which triggers an execution step in the functional process (for example with *sync_pw*1 for triggering *Sensor*1 as shown Fig. 4).

The assessment of the value *enable_tic* is set to *true* only if the *clock_state* value is equal to 2. In other cases, *enable_tic* are set to $false$. The value *dead* is set at *true* when the associated clock should not be active in the rest of the execution.

The generation automatically produces the *Scheduler* code including this part executed during the *Active* step:

```
... if (tab_Clocks [0].enable_tick) then sync_pw1
elsif (tab_Clocks [1].enable_tick) then sync_pr1
elsif (tab_Clocks [2].enable_tick) then sync_pw2
elsif (tab_Clocks [3].enable_tick) then sync_pr2
elsif (tab_Clocks [4].enable_tick) then sync_comput
elsif (tab_Clocks [5].enable_tick) then sync_filter (true)
elsif (tab_Clocks [6].enable_tick) then sync_filter (false)
end ...
```

Translation of Constraints: We implement each CCSL constraint by a Fiacre process that implements the automaton (cf Section 3.1) corresponding to the constraint (we called those processes *constraints processes*). These process are synchronized with *Scheduler* via the port *start*, *update* and *end* for the activation of automaton transitions. For example, we show the code for the $alternatesWith$ constraint corresponding to the automaton shown in Fig. 2.b. The transitions of this automaton are triggered by signal ports *startA*, *updateA* and *endA* and update the value of *clock_state*. The encoding principle for the other two constraints, strict precedence and filtering is similar.

```
process AlternatesWith [startA, updateA, endA: in none] // ports
  (&c1: nat, &c2 : nat, &tab_Clocks: T_ARRAY_CLOCK)    // shared variables
is states s1, s2, s3, s4, s5
init to s0
from s0  startA;
  tab_Clocks [c1].clock_state := 2; tab_Clocks [c2].clock_state := 1; to s1
from s1  updateA;  to s2
from s2  endA; to s3
from s3  startA;
  tab_Clocks [c1].clock_state := 1; tab_Clocks [c2].clock_state := 2; to s4
from s4 updateA;  to s5
from s5 endA; to s0
```

6 Formal Verification of Properties

6.1 Verification Principles

To verify a set of requirements on a model, we must explore it exhaustively and have a formal expression of properties to be checked, for example in the form of logical formulas or observer automata. In our approach, we express the properties with CDL language.

Once the observers have been specified, the model is then explored and the exploration generates a labeled transition system (LTS). It represents all the behaviors of the model in its environment as a graph of configurations and transitions. On this LTS, the verification of the properties is carried out by applying a reachability analysis of observer error states.

6.2 Expressing Properties Using CDL

The CDL language allows the user to specify properties which are expressed as predicates or observer automata. Predicates in CDL reference variables values: for example, *predicate pred*1 *is* $\{\{Proc\}1 : v = value\}$ means *pred*1 is true if the variable v of the first instance of the $Proc$ process is equal to the value *value*. A predicate can also reference a process state: for example, *predicate pred*2 *is* $\{\{Proc\}1@stateX\}$ means that *pred*2 is true if the first instance of the $Proc$ process is in the state $stateX$. A predicate can also reference the amount of data contained in a $fifo$ or a boolean expression combining the previous types of predicates.

This syntax provides a rich mode of expression that together with the observer, enables the expression of properties which would be difficult to express in linear logic (see the P2 observer Fig. 7). The predicates allow insights into the behavior of a model while providing expression which is easy to use and understand for the designer. In our work, we express properties in CDL following two complementary objectives: one to verify that the implementation of CCSL constraints is correct, the other to ensure that the functional parts of the circuit ($Sensor1$, $Sensor2$, $Acq1$, $Acq2$, $Comput$, $Filter$) are properly implemented.

Properties Associated with CCSL Constraints: Here we illustrate the specifications of some properties associated with CCSL constraints included in our system model. The goal is to prove the correct Fiacre implementation of Scheduler and constraint automata. To check a property $P1$ associated with the alternation requirement $Req1$, for example $write1$ $alternatesWith$ $read1$, we declare the CDL events evt_write1 and evt_read1 (Fig. 6.a). With these events, we specify the observer, illustrated in Fig. 6.b), encoding the property $P1$ which satisfies the alternating synchronization $write1$ and $read1$. The initial state of the observer is the *Start* state and has an error state (*Reject*). Each transition of the observer is triggered by the occurrence of an event (evt_write1 or evt_read1).

In a similar way, we can specify observers to verify properties of the requirement $Req2$ by declaring the events evt_read_2 and evt_comput:

```
event evt_read2  is  {sync sync_pr1 from {Scheduler}1 to {Sensor}2}
event evt_comput is  {sync sync_comput from {Scheduler}1 to {Comput}1}
```

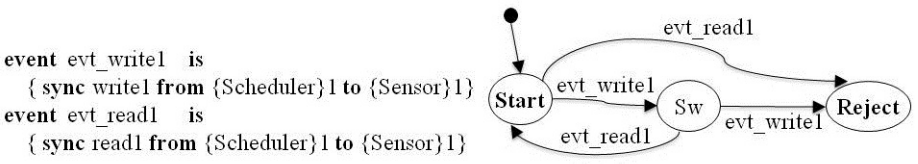

Fig. 6. Observer automaton corresponding to P1 property

The CDL language also allows to specify predicates that can be verified during the exploration of the model. For example, if we want to check that, in an instant, clocks $write1$ and $read1$ do not "tick" at the same instant, we can declare the following predicates:

```
predicate enable_tick_pw1_true is {{C}1:tab_Clocks [0].enable_tick = true}
predicate enable_tick_pr1_true is {{C}1:tab_Clocks [1].enable_tick = true}
predicate enable_tick_rw1_together is
                  {enable_tick_pw1_true and enable_tick_pr1_true}
```

We can now declare, with the operator $assert$[5], the following invariant: $not\ act\_tick\_rw1\_together$. During the exploration of the model, the OBP tool checks that the invariant is not violated.

The CDL predicates can also facilitate the writing of more complex observers when they refer to a large number of events. For example, the requirement $Req3$ associated with the generation of $data$ by $Comput$ and the filtering constraint is expressed by the CCSL term: $filterOut = comput\ filteredBy\ (001)^w$. During the exploration, we need to verify that the sequence of data generated from $Filter$ is the sequence generated by $Comput$ with a sampling of one value in 3. In the current version of the model, the filter word (001) is stored in an array variable $tabFilter$ of the constraint process $FilteredBy$. The $(i\ modulo\ 3)^{th}$ datum of the sequence generated by $Comput$ will be copied in the sequence derived from $Filter$ if the value $tabFilter[i\ modulo\ 3]$ is equal to 1. Otherwise, it is not copied into the sequence of data supplied to the environment. To verify this constraint, we therefore declare the following predicates (for $x \in \{0, 1, 2\}$):

```
predicate bitx_true  is {{FilteredBy}1:tabFilter[x] = 1}
predicate bitx_false is {{FilteredBy}1:tabFilter[x] = 0}
```

Transitions of an observer can be decorated with one of the predicates together with the events evt_comput, $evt\_filterTrue$ and $evt\_filterFalse$ which trigger the transitions of the observer and they are declared as follows:

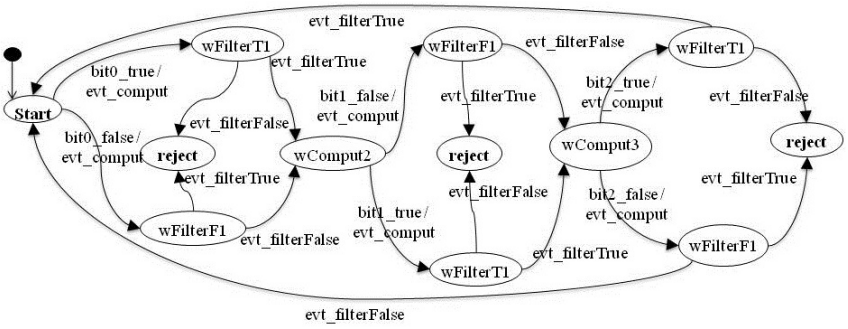

Fig. 7. Observer automaton corresponding to the P2 property

[5] See detailed syntax of the CDL language available at `http://www.obpcdl.org`

```
event evt_filterTrue  is {sync filter (true)  from {Scheduler}1 to {Filter}1}
event evt_filterFalse is {sync filter (false) from {Scheduler}1 to {Filter}1}
```

Figure 7 illustrates the observer encoding property $P2$ and referencing the above predicates and events.

If we want to verify other properties on the functional parts of our model, we specify these properties which characterize the behavior of the model. For example, the $Req4$ requirement, expressed in Section 4, can be expressed by an observer automaton using predicates and appropriate events.

7 Experimentation on the Case Study and Discussion

To conduct the experiments, we implemented the OBP tool (Fig. 1). OBP is structured in three modules. The *front end* OBP imports Fiacre models corresponding to a translation of UML-MARTE models including CCSL specifications. In addition, it imports CDL programs which describe the properties and context scenarios if required. *OBP Explorer* explores the model, and after each transition model run, it hands over to the *Observation Engine*. It captures the occurrences of events and evaluates, at each step of the running model, the value of predicates and the status of all involved observers. A verification of all invariants and reachability analysis of error state observers is thus conducted.

At the end of exploration, a report is generated by OBP, revealing the list of property evaluated to true or false. Also, OBP provides either counter examples on request on the *reject* or *success* observer state accessibility or invariant violations. These indications may refer the user to the scenario having the defeated properties. We are currently working to facilitate the interpretation of data provided by OBP and to display understandable data in the user's models, allowing ease of diagnosis.

With CDL, we specified observers to verify different properties concerning requirements ($Req1$ to $Req4$) expressed in Section 4. For this proposed case study, the complexity of exploration[6] is reasonable size. As an example, if we assume that the size of fifo is equal to 1, the number of explored configurations is then 45 040 and the number of transitions is 167 496. If the size is equal to 3, the number of explored configurations is then 359 104 and the number of transitions is 1 702 704.

The use of the Fiacre language in our translation approach serves to reduce the semantic gap between high-level models expressed in UML MARTE description, by making it possible to precisely specify the semantics of the input language for system modeling. This intermediate language enables to share these specifications through different verification tool-chains. Our CDL language can be compared with the *Property Specification Language* (PSL) [IEE05]. In future work, we investigate to compare CDL expressiveness with PSL and the discussion in [Mal12] is very interesting for this topic.

[6] The tests are run on a machine such as Windows 7, 64-bit - 4 GB RAM with OBP v.1.3.4.

8 Conclusion

In this work, we have presented an implementation of CCSL constraints in the Fiacre language and we expressed properties in the CDL language. The manipulation of CCSL expressions within the framework of modeling with UML-MARTE formalism can extend the expressiveness by integrating temporal constraints into the model. The logical time model proposed by the OMG to enrich the UML MARTE allows the description and analysis of temporal constraints. We have defined a automatic translation approach to generate Fiacre programs from UML-MARTE models enriched with CCSL constraints. This approach allows to verify formally the implementation of CCSL constraints and functional requirements.

We carried out a verification technique of properties by model-checking using the CDL language and the OBP tool. CDL can easily express predicates and observers which are checked during the exhaustive model exploration by OBP. We have shown that this language facilitates the expression of properties. They can be expressed with a very fine granularity, referencing variables and process states.

We can take benefits of the CCSL automata encoding. These automata are as reusable inputs to apply the verification. Our translation approach can be an important step toward the formal verification process of both MARTE models and CCSL specifications. Once the translation of CCSL constraints into Fiacre is complete, the operation requires only a single verification as it does not depend on the modeled application. Even though the model may change, the Fiacre code is reusable as this translation principle is independent of the application. We think that our approach contributes to clarify its role when addressing this domain by expressing temporal properties dedicated to CCSL relation constraints.

Acknowledgment. We wish to thank Dr Zoé Drey for her valuable and constructive suggestions related to this paper.

References

[And09] André, C.: Syntax and semantics of the clock constraint specification language ccsl. Technical Report 6925, INRIA (2009)

[And10] André, C.: Verification of clock constraints: Ccsl observers in esterel. Technical Report 7211, INRIA (2010)

[CES86] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)

[DBRL12] Dhaussy, P., Boniol, F., Roger, J.-C., Leroux, L.: Improving model checking with context modelling. In: Advances in Software Engineering, ID 547157, 13 pages (2012)

[DMA08] DeAntoni, J., Mallet, F., André, C.: Timesquare: on the formal execution of uml and dsl models. In: Tool Session of the 4th Model Driven Development for Distributed Real Time Systems (2008)

[FGP+08] Farail, P., Gaufillet, P., Peres, F., Bodeveix, J.-P., Filali, M., Berthomieu, B., Rodrigo, S., Vernadat, F., Garavel, H., Lang, F.: FIACRE: an intermediate language for model verification in the TOPCASED environment. In: European Congress on Embedded Real-Time Software (ERTS), Toulouse. SEE (January 2008)

[HLR93] Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous observers and the verification of reactive systems. In: Nivat, M., Rattray, C., Rus, T., Scollo, G. (eds.) Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST 1993, Twente. Workshops in Computing, pp. 83–96. Springer Verlag (June 1993)

[IEE05] IEEE. IEEE standard for property specification language (psl). Technical Report 1850 (2005)

[Mal12] Mallet, F.: Automatic Generation of Observers from MARTE/CCSL. In: RSP 2012 - International Symposium on Rapid System Prototyping, Tampere, Finlande, pp. 86–92. IEEE (October 2012)

[MAS08] Mallet, F., André, C., De Simone, R.: Ccsl: Specifying clock constraints with uml/marte. ISSE 4, 309–314 (2008)

[OMG10] OMG. Uml profile for marte, v1.1. Object Managment Group, Document number: PTC/10-08-32 (August 2010)

[QS82] Queille, J.-P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)

[YM11] Yin, L., Mallet, F.: Correct transformation from ccsl to promela for verification. Technical Report 7491, INRIA (2011)

[YTB+11] Yu, H., Talpin, J.-P., Besnard, L., Gautier, T., Marchand, H., Le Guernic, P.: Polychronous controller synthesis from marte ccsl timing specifications. In: Memocode (2011)